Compressed Inverted Indexes for In-Memory Search Engines

Frederik Transier* Peter Sanders†

Abstract

We present the algorithmic core of a full text data base that allows fast Boolean queries, phrase queries, and document reporting using less space than the input text. The system uses a carefully choreographed combination of classical data compression techniques and inverted index based search data structures. It outperforms suffix array based techniques for all the above operations for real world (natural language) texts.

1 Introduction

Searching in large text data bases has become a key application of computers. Traditionally (a part of) the index data structure is held in main memory while the texts themselves are held on disks. However, this limits the speed of text access. Therefore, with ever increasing RAM capacities, there is now considerable interest in data bases that keep everything in main memory. For example, the TREX engine of SAP stores large quantities of small texts (like memos or product descriptions) and requires rapid access to all the data. Such a search engine consists of a cluster of multi-core machines, where each processing core is assigned an about equal share of the data base. Since RAM is hundreds of times more expensive than disks, good data compression is very important for in-memory data bases. In recent years, the algorithms community has developed sophisticated data structures based on suffix arrays that provide asymptotically efficient search using very little space that can even be less than the text itself.

This paper studies a different approach to compressed in-memory text search engines based on inverted indexes. We show that with careful data compression we can use very little space. The system consists of several interacting parts that support Boolean queries, phrase queries, and document reporting. It turns out that the parts interact in a nontrivial way. For example, the index data structure introduced in [1] turns out to allow better data compression as a side effect and can be used to simplify the positional index used for phrase queries. Both indexes together are used for a fast document reporting functionality that does not need to store the original text. Since most previous papers focus only on a subset of these aspects, we believe that our results are of some interest both from the application perspective and for algorithm engineering.

We therefore compare our framework with publicly available implementations based on suffix arrays. Somewhat unexpectedly, our system is more space efficient than compressed suffix arrays. Its query performance is up to 3–4 orders of magnitude faster for Boolean queries, about 20 times faster for phrase queries (the application domain for which suffix arrays are designed), and about five times faster for document reporting.[1]

We give a more concrete definition of the key terms in Section 2 and related work in Section 3. Section 4 with our inverted index data structures constitutes the main algorithmic part of the paper. We call it compressed inverted index. Its modular design consists of many ingredients each of which is given by a well defined software module whose implementations can be exchanged. In Section 5 we explain how to implement all required operations using compressed suffix arrays. Section 6 gives an experimental evaluation using real world benchmark data and Section 7 summarizes the results and outlines possible future work.

2 Preliminaries

We consider the problem of querying for a subset of a large collection of *normalized* text documents. A normalized text document is a set of words or terms obtained from an arbitrary text source by smoothing out all so-called term separators, i.e. spaces, commas and so on. Thereby we do not distinguish between upper and lower-case characters. For simplicity, we map each term and each document to an integer value. So sometimes we refer to documents or terms as *document identifiers (IDs)* or *term IDs* respectively.

*SAP AG (NW EIM TREX), 69190 Walldorf, Germany and Universität Karlsruhe (TH), 76128 Karlsruhe, Germany - transier@ira.uka.de

†Universität Karlsruhe (TH), 76128 Karlsruhe, Germany - sanders@ira.uka.de

[1] Note that we do not make any claims about the performance of suffix arrays for other applications such as in bioinformatics where the data has very different structure and where we also need different operations.

A few types of queries are of particular importance: A Boolean *AND* query on a document collection asks for all documents that contain all elements of a given list of terms. Analogously, a Boolean *OR* query searches for all documents that contain any of the terms in the list. And finally, a *phrase* query asks for documents that contain a given sequence of terms.

In a real-world search engine we are obviously not interested in a list of document IDs as the result set. Rather, we expect the document texts itself. So we consider the reporting of result documents as an important functionality of text search engines as well.

The most widely used data structure for text search engines is the *inverted index*. In an inverted index, there is a *inverted list* of document IDs for each term ID showing in which document a term appears. In this simplest version of an inverted index, it can just answer Boolean queries. For phrase queries additional positional information is needed, i.e. for each term in a document a list of positions where it occurs. We call indexes that contain such information *positional indexes*.

Of course, there are many alternatives to the inverted index in the literature. Some of them have similar but somewhat different fields of application. So *suffix arrays* are mainly used for substring queries. They store all suffixes of a given text in alphabetical order. In this way, a substring pattern can quickly be found using binary search. And since a while ago, there are also compressed versions of suffix arrays.

Just as well, compression can be applied to inverted indexes. There exists a couple of standard compression techniques suitable for inverted lists. A straight forward approach is *bit-compression*. For each value in the list, it uses the number of bits needed to code the largest one among them. Thus, further compression can be achived by trying to reduce the maximum entry of a list. For example, a sorted list can be transformed into a list of differences between every two consecutive values. These *delta* values are smaller or at least equal to their origins and thus, they can probably be stored using fewer bits. The big drawback of those *linear* encoding schemes is that a list have to be unpacked from the front for accessing any value in it.

There are also variable bit length encodings. A very popular one is the *Golomb* encoding from [2]. It is often used on delta sequences. Golomb coding has a tunable parameter. A value is coded in two parts. The result of a division by the parameter and the remainder. The quotient is coded in unary followed by the remainder in truncated binary coding.

3 Related Work

Data structures for text search engines have been studied extensively in the past. Zobel and Moffat [3] give a good overview. Across the relevant literature, the inverted index structure has been proved to be the method of choice for fast Boolean querying. As a consequence, a lot of work has been done in order to improve the basic data structure of a single inverted list with regard to compression and simple operations such as intersections or unions (q.v. [4, 1]).

Traditionally, search engines store at least the major parts of their inverted index structures on hard disks. They use either storage managements that are similar to those of Zobel et al. [5] or the persistent object stores of databases (cp. [6]). Due to the growing amount of random access memory in computer systems, recent work have considered to purge disk-based structures from index design. Strohman and Croft [7] show how *top-k* Boolean query evaluation can be significantly improved in terms of throughput by holding impact-sorted inverted index structures in main memory. Besides strict in-memory and disk-based solutions there are some hybrid approaches. Luk and Lam [8] propose a main memory storage allocation scheme suitable for inverted lists of smaller size using a linked list data structure. The scheme uses variable sized nodes and therefore well suited for inverted lists that are subject of frequent updates. Typically, such approaches are used in combination with larger *main* indexes to hide the bottleneck of updating search optimized index structures (cp. [9]).

Unbeaten for Boolean queries, there is a rub in querying for phrases within inverted indexes. Those queries are slow as the involved position lists have to be merged. Several investigations have been made for reducing this bottleneck. A simple approach of Gutwin et al. [10] is to index phrases instead of terms. So-called *nextword indexes* were proposed by Bahle et al. [11]. For each term, they store a list of all successors together with their corresponding positions. Furthermore, a combined index approach of the previous ones was proposed by Williams et al. [12].

Suffix trees and suffix arrays were originally designed for substring search. Hence they can also be used for efficient phrase querying. Due to their high space consumption, they were first unsuitable for large text indexing. However, by now, various authors have worked on the compression or succinct representations of suffix arrays. For an overview see Navarro and Mäkinen [13] or the Pizza&Chili Website[2]. According to González and Navarro [14], they need about 0.3 to 1.5 times of

[2] http://pizzachili.di.unipi.it/

the size of the indexed text — depending on the desired space-time trade-off. Moreover, they are *self-indexing*, i.e. they can reconstruct the indexed text efficiently. In comparison to that, an inverted index needs about 0.2 to 1.0 times of the text size ([15]) depending on compression, speed and their support of phrase searches. Unfortunately, inverted indexes can not reconstruct the text, so they require this amount of space in addition to the (compressed) text size.

In recent work, Ferragina and Fischer [16] investigated in building suffix arrays on terms. They need just about 0.8 times of the space of character based suffix arrays being twice as fast during the construction.

Puglisi et al. [17] compared compressed suffix arrays and inverted indexes as approaches for substring search. They indexed q-grams, i.e. strings of length q, instead of words to allow full substring searches rather than just simple term searches within their inverted index. As their result, they found out that inverted indexes are faster in reporting the location of substring patterns when the number of their occurrences is high. For rare patterns they noticed that suffix arrays outperform inverted indexes. Bast et al. have observed that suffix arrays are slower for prefix search [18] and for phrase search [19].

4 Compressed Inverted Index

The core of our text index is a document-grained inverted index. For each term ID we store either a pointer to an inverted list or — if the term occurs in one single document only — just a document ID. We distinguish between two types of inverted lists. The first type is used for all terms that occur in less than $K = 128$ documents. In our implementation we use the delta Golomb encoding in order to compress these *small* lists in a simple but compact way. For the remaining terms, we use an extension to the two-level lookup data structure of [1]. It is organized as follows. Let $1..U$ be the range of the N document IDs to be stored in a list. Then the list can be split into k buckets spanning the ranges $u_i = (i-1)\frac{U}{k}..i\frac{U}{k}$ with $i = 1..k$. According to [1], the parameter k is thereby chosen as $\lceil \frac{N}{B} \rceil$ with $B = 8$, so that the buckets can be indexed by the $\lceil \log k \rceil$ most significant bits of their contents. The top-level of the data structure stores a pointer to each bucket in the bottom-level. For accessing the positional index, we also need a *rank* information, i.e. $\sum_{j=0}^{i} n_j$ for each bucket i whereas n_j denotes the size of bucket j. An interesting side effect is that using the rank information, we can guess the average delta within each bucket i as the quotient of the range delta $\frac{U}{k}$ and the rank delta n_i. In this way, we can choose a suitable Golomb parameter for a *bucket-wise* delta encoding according to [15] as $b_i = \ln(2)\frac{U}{kn_i}$.[3] Obviously, the initial delta value of the i-th bucket is given by $i^{\lceil \log \frac{U}{k} \rceil}$. In contrast to the linear encoding of the buckets, all top-level values are required to be randomly accessible. The top-level is later used to truncate the considered data (cp. algorithm lookup in [1]) and hence, to reduce the querying time. We call this two-level data structure a *large* inverted list.

The distinction between single values, small and large lists offers two advantages. On the one hand, we can store lists smaller than K in a very compact way. On the other hand, this helps to avoid scanning huge lists.

LEMMA 4.1. *Assuming the classical Zipfean distribution of M terms over N documents, our index contains at least $M - \frac{N}{\ln(M)}$ single value entries.*

Proof. According to [20], the classical Zipfean law says that the rth most frequent term of a dictionary of M words occurs $\frac{N}{rH_M}$ times in a collection of N documents, where H_M is the harmonic number of M. Claiming an occurrence frequency of 1, we get $r_1 = \frac{N}{H_M}$ as the order of the first term that occurs just once. As the total number of terms in our index is M, we have $M - \frac{N}{H_M}$ unique terms. Due to [21] holds $H_n - \ln(n) - \gamma < \frac{1}{2n}$, where γ is the Euler-Mascheroni constant. Therefore, $M - \frac{N}{H_M} > M - N(\frac{1}{2M} + \ln(M) + \gamma)^{-1} > M - \frac{N}{\ln(M)}$.□

LEMMA 4.2. *Our document-grained index contains at least $(1 - \frac{1}{K})\frac{N}{\ln(M)}$ small lists.*

Proof. As in proof of Lemma 4.1, we can obtain the order of the first term that's occurrence frequency is equal to K using the Zipfean law, i.e. $r_K = \frac{N}{KH_M}$. By subtracting this rank from that one of the unique terms r_1, we get the number of terms that occur more than once but less or equal than K times. Thus, $r_1 - r_K = (1 - \frac{1}{K})\frac{N}{H_M}$ and due to [21] $r_1 - r_k > N(1 - \frac{1}{K})(\frac{1}{2M} + \ln(M) + \gamma)^{-1} > (1 - \frac{1}{K})\frac{N}{\ln(M)}$. □

As an extension of our document-grained inverted index we store the positional information of each term in a separate structure. Again, we distinguish the way of storing the data depending on the list lengths: A vector indexed by the term IDs provides the basis. For all terms that occur only once within the index, we store the single position itself. For the remaining terms we store pointers to lists of different types. We use simple linear encoded lists for terms that occur more often but in just one single document. Terms that appear in multiple documents just once in each, are stored in a

[3] In our experiments, this saves about 35 % space compared to using a global Golomb parameter.

random accessible list, i.e. using bit-compression. And finally, for all the others, we use *indexed lists*. An indexed list is a two-level data structure whose top-level points to buckets in the bottom level containing an arbitrary number of values each. In our case, a bucket contains a list of positions of the term within a certain document. It is indexed from the top-level by the rank of that document in the document-grained index. Figure 1 shows how to retrieve term positions from indexed lists for a given term v and a list of document ID-rank pairs D.

```
Function positions(v, D)
    O := ⟨⟩                                  // output
    foreach (d, r) ∈ D do                    // doc ID - rank pairs
        i := t^v[r]                          // start of position values
        e := t^v[r + 1]                      // end of position values
        while i < e do                       // traverse bucket
            O := O ∪ (d, b^v[i])             // add to result
            i++
    return O
```

Figure 1: High level pseudocode for retrieving positional information.

In a similar way, we have access to the positional information coded in the bit-compressed lists. We can consider them as lists that consist of a top-level only. However, instead of storing pointers to buckets therein, we store the single position values directly. Defining a function *positions* for these lists is trivial. And for the terms that occur in just one document it is even easier.

The advantages of the separation between document-grained and positional information are obvious: For cases in which we do not need the functionality of a positional index, we can easily switch off this part without any influence on the document-grained data structures. Furthermore, Boolean queries are very cache efficient as the inverted lists are free of other information than the document IDs itself. And even phrase queries are supposed to be fast. Due to the fact that we work in main memory we can jump to the positional information *actually* needed at little cost.

As usual for inverted index based text search engines we use a dictionary that maps normalized terms to term IDs and vice versa. Since this part of a search engine is not our main focus, we use a simple uncompressed dictionary. It stores the normalized terms alphabetically ordered and uses binary search for retrieving IDs for a given term. The inverse mapping is done via a separate vector that is indexed by the term IDs and contains pointers to the respective terms. Surely, our dictionary has not its strengths in mapping terms to term IDs very fast, but as it holds uncompressed terms, it has certainly no harmful influence on our reconstruction process in Section 4.2.

Memory management turned out to be crucial for obtaining *actual* space consumption close to what one would expect from summing the sizes of the many ingredients. Most data is stored in blocks of 512 Kbytes each. Lists that are greater than such a block get their own contiguous memory snippet. Within the blocks, we use word aligned allocation of different objects. We tried byte-alignment but did not find the space saving worth the computational overhead.

4.1 Querying the Compressed Inverted Index

Boolean Queries. Inverted indexes are by design well suited to answer Boolean queries. The *AND* query corresponds to an intersection of a set of inverted lists. This problem has been studied extensively by various authors in the past (cp. [22, 4, 1]). For our index we use a simple binary merge algorithm for all small lists and algorithm *lookup* from [1] for large lists.

Phrase Queries. The phrase search algorithm on the index can be divided into four steps. First, we sort the phrase terms according to their frequency, i.e. their inverted list lengths. Of course, we have to bear in mind the term positions within the phrase. Then, we intersect the document-grained inverted lists of the two least frequent terms — using algorithm *lookup* if applicable — keeping track of the current rank for each list. With this information, we can retrieve the corresponding position lists using function *positions* of Figure 1. As a result, we have two lists of pairs consisting of a document and the position within the document. In a third step, we combine the two lists using a simple binary merge algorithm that normalizes the positions according to the query phrase on the fly. Finally, the further terms have to be incorporated into the result set. In increasing order of frequency, we repeat the following for each term: First, we intersect a terms inverted list with the current result set. Then, we retrieve the positions. And in the end, we merge them with the previous result set.

4.2 Document Reporting. The query result of the inverted index is a list of document IDs which are associated with pointers to the original documents. Traditionally, the documents are archived in a separate storage location, i.e. in files on disk or on the network. They were retrieved from there when the search engine returns the results. However, since we want to exploit

associated advantages of main memory, our space is scarce. So instead of storing the documents separately, we use the information about the indexed documents we have already stored. In fact, we know the term ID - document ID mappings from the inverted lists, as well as the position lists for these ID pairs. So we could restore the sequence of normalized terms of a document by traversing the inverted lists, gathering the term IDs and placing their dictionary items using algorithm *positions*. However, reconstructing a term sequence this way would take too much time. Instead, we store a *bag of words* for each document in addition to our existing data structures. A bag of words is a set of all the term IDs occurring in a document. We store them sorted in increasing order of IDs using delta Golomb encoding. Besides, we encode sequences of consecutive value IDs as a 0 followed by the size of the series. So we are able to build term sequences without traversing through all inverted lists, just by positioning all terms of the bag.

There is still a small step from the sequence of normalized terms towards the original document we have indexed before. For that reason, we store the changes made to the terms during the normalization process. In our dictionary, all terms are normalized to lower-case characters — the most frequent spelling of words in English texts. Any differences to this notation in the original document are recorded in a list of *term escapes*. An item in that list consists of a pair of a position where the variation in the sequence occurs and an escape value. The escape value indicates how the term has to be modified. A value of 0 means that the first character has to be capitalized and a value of 1 means that the whole term has to be spelled with capital letters. Each greater escape value points to a replacement term in a separate *escape dictionary*.

It remains to incorporate the term separators into the string of adjusted terms. We store the arrangement of separators and terms of each document as a (Golomb-coded) list of values: A 0-value followed by a number x indicates that there are x terms separated by spaces. A 1-value indicates a term (Recall that the normalized text already tells us *which* term). A value of 2 or larger encodes the ID of a separator. We use a simple binary merge algorithm to reconstuct a document from its term sequence and its separator arrangement.

5 Document Retrieval on Compressed Suffix Arrays

In order to qualify our phrase querying performance, we have implemented document retrieval algorithms on compressed suffix arrays. Of course, suffix arrays know neither terms nor term IDs, so we do not need a dictionary here. However, as we expect the same query results, we perform the same normalization process to obtain normalized documents, i.e. normalized document terms separated by spaces. We concatenate all documents separated by a special end of document character into single text from which we build the suffix array.

As the suffix array knows global character positions only, we need to store all document starting positions in a separate data structure. Again, we use a *lookup* list as described in section 4, in which the rank information correspond to the document IDs.

5.1 Querying Suffix Arrays

Phrase Queries. Phrase querying on suffix arrays is straight forward. Here, a phrase search is equal to a substring search of the normalized query phrase. Due to the special separator characters, it is impossible to get a result phrase that overlaps document bounds. As the result, the suffix array returns positions where the phrase occurrences start. They have to be remapped to document IDs using the lookup list described above.

Boolean Queries. For Boolean queries, we first locate the occurrences for each query term by a substring search in the suffix array. Afterwards, we map the result list of the least frequently occurring term to a list of document IDs. Then, we insert the list in a binary search tree (`std::map`) and incorporate the results of the next frequently occurring term by performing a tree lookup for each of its items. We put all hits in a second tree and swap them as soon as we have checked all of the items. This process is repeated until all term IDs are processed.

We also tried two other approaches for Boolean queries. The first one was to build a hash table from the shortest document ID list using `stdext::hash_map` and to check there for the IDs of the remaining lists. The second one was to sort all lists and to intersect them like the inverted lists. However, these two alternatives could not compete with the previous one. Anyway, the major parts of the querying time are spent during suffix array operations. We will see this in the experimental section.

5.2 Document reporting. Document reporting on suffix arrays is simple. They support this operation innately. However, the document bounds required for reporting have to be retrieved outside the suffix array in our lookup data structure.

Of course, as we have indexed the normalized text into the suffix array, it returns the normalized text as well. In order to get the original input document, we

have to take a similar process as we used while retrieving original documents from our inverted index. We did not implement this, so we will compare just the reconstruction of the normalized text in our experimental section.

6 Experiments

Table 1: Index properties (50 000 documents of WT2g)

	CII	CSA
dictionary	23.9	-
document index	32.3	-
positional index	126.3	-
bag of words	23.7	-
suffix array	-	230.8
doc bound list	-	0.1
sum [MB]	**206.1**	**230.9**
text delta	108.7	-
	314.8	-
input size (norm.)	412.8 (360.5)	- (360.5)
compression	0.76 (0.57)	- (0.64)
indexing time [min]	**5.6 (5.1)**	**- (9.3)**
peak mem usage [GB]	0.7	3.2

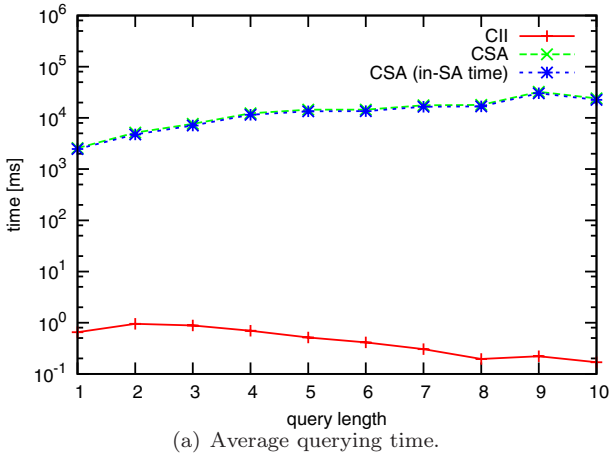
We have implemented all algorithms using C++. We used a framework wich allows us to switch easily between different approaches while preserving the environment for our measurements. Most tuning parameters in our implementation were set heuristically using back-of-the-envelope calculations and occasional experiments. We have not spent much effort in systematic tuning or extensive evaluation of tradeoffs since good space-time compromises are usually easy to obtain that are not very sensitive with respect to the values chosen.

The experiments were done on one core of an Intel Core 2 Duo E6600 processor clocked at 2.4 GHz with 4 GB main memory and 2 × 2048 Kbyte L2 cache, running OpenSuSE 10.2 (kernel 2.6.18). The program was compiled by the gnu C++ compiler 4.1.2 using optimization level -O3. Timing was done using PAPI 3.5.0[4].

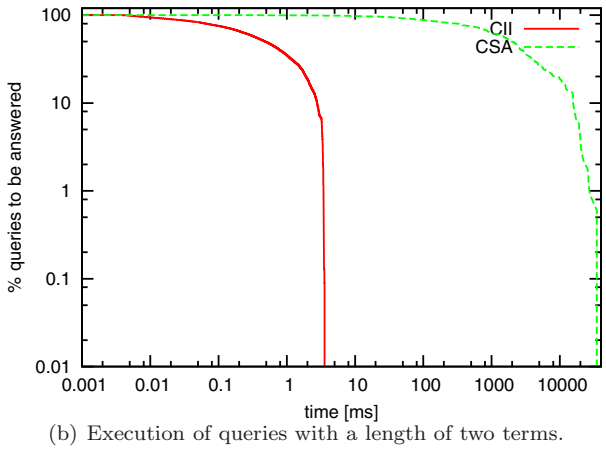
For our experiments we used the WT2g[5] text corpus, which is a standard benchmark for web search. Although WT2g is by now considered a 'small' input, it is right size for the amount of data assigned to *one processing core* of an *in-memory* text search engine based on a cluster of low cost machines. Using more than 1–2 GByte of data on one core would mean investing much more money into RAM than into processing power.

[4] http://icl.cs.utk.edu/papi/

[5] http://ir.dcs.gla.ac.uk/test_collections/access_to_data.html



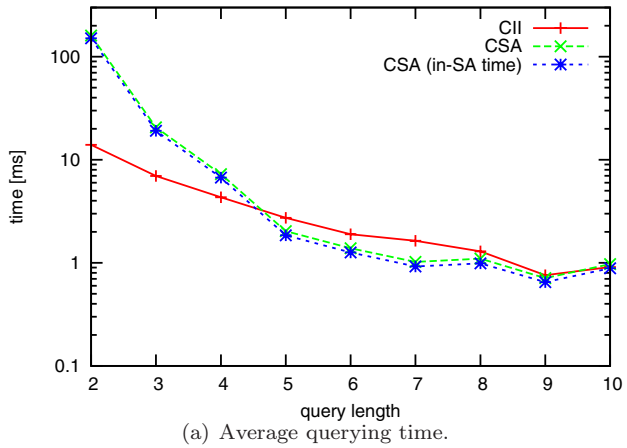
(a) Average querying time.



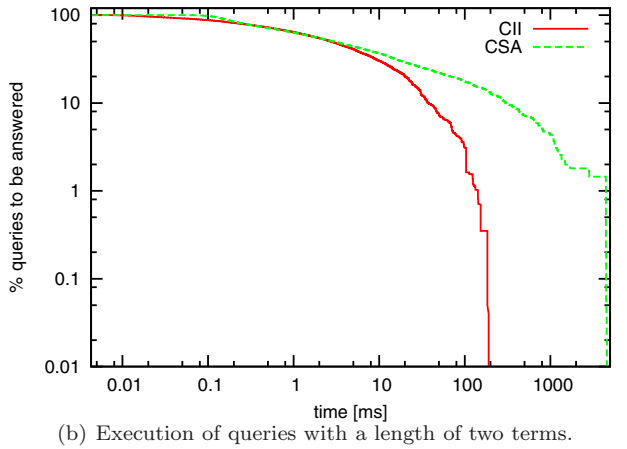
(b) Execution of queries with a length of two terms.

Figure 2: *AND* queries on the first 50 000 documents of WT2g.

As a representative of the compressed suffix indexes, we used the compressed suffix array (CSA) implementation by Sadakane available from the Pizza&Chili website. Compressed suffix arrays showed the best performance among the highly space efficient implementations on our system. For a comparision of further Pizza&Chili indexes based on a WT2g subset see Appendix A. We built the suffix array via the API using default parameters. The peak memory usage for indexing the first 50 000 documents of WT2g into a compressed suffix array was already at the limits of our physical main memory. So the following comparison will be based on this subset. Additionaly, we have conducted our experiments for the compressed inverted index on the complete WT2g corpus, as well as on WT2g.s which was already used in [1]. It is derived from the former by indexing each sentence of the plain normalized text as a single document. The results are shown in Appendix B.



(a) Average querying time.



(b) Execution of queries with a length of two terms.

Figure 3: *Phrase* queries on the first 50 000 documents of WT2g.

Table 1 contains the properties of the two different indexes. The size of the parts of the compressed inverted index (CII) that provide the same functionality as the CSA is about 10% smaller than the CSA index and about 43% smaller than the input text. In addition, 26% of the input size is required by the CII for reconstructing the original documents from their normalized versions. This still leaves a saving of more than 20% compared to the input text.

The indexing time of both methods is comparable but ours needs only about one fifth of the memory during construction. We still need about twice as much memory than the input text during compression. However, note that the text itself could be streamed from disk during construction. Furthermore, we can build the index for one core at a time (possibly in parallel), temporarily using the space of the other cores for the construction. Hence, this looks like an acceptable space consumption.

For evaluating the querying performance, we generated pseudo real-world queries by selecting random hits. From a randomly chosen document, we used between one and ten arbitrary terms to build an *AND* query. Similarly, we chose a random term from such a document as the start of a *phrase* query. The lengths of the generated phrase queries ranged between two and ten terms. We built 100 000 queries varying in their lengths according to the distribution given in [23], where it is reported that over 80% of real world queries consists of between one and three terms.

Our first experiment was to determine the *AND* querying performance of the two approaches. Figure 2(a) shows the average query time over the first 10 000 of our generated queries. As expected, the inverted index performs well over the entire range of query lengths. It benefits from a larger number of query terms since this makes it more likely that the query contains a rare term with a short inverted list. Since the running time of the intersection algorithm [1] is essentially linear in the smaller list, this decreases processing time. In contrast, the compressed suffix arrays produce unsorted lists of occurrences for all query terms that have to be generated using complicated algorithms which cause many cache faults. This takes the major part of the querying time (in-SA time). In comparison to that, the time required for mapping the positions to document IDs and merging the lists is negligible. The bottom line difference is huge. On the average, our data structure is more than 5 000 times faster than CSA. In Figure 2(b) we took a closer look at the 'difficult' queries with a length of two terms. The figure shows how many percent of the queries take longer than a given time. In a double logarithmic plot, both curves have a quite clear cutoff. However, the differences are again huge. While inverted indexes never took more than 3.6 ms, the CSA needs up to 35 s, almost 10 000 times longer. We can conclude that a search engine based on suffix array would probably need an additional inverted index at least for the Boolean queries.

Our next experiment was to investigate the *phrase* querying performance — a home match for suffix arrays. In Figure 3(a) we see the average time required for the 100 000 queries of our pseudo real-world query set. For the most frequent practical case of two terms, the inverted index is nevertheless more than 20 times faster. Suffix arrays are slightly faster for the rare and easy phrases with more than four terms. The distribution of the query times in Figure 3(b) indicates that the largest observed query times are a factor 24 apart. CSA needs nearly 5s for some of the phrases. The reason is that some phrases occur very frequently, and unpacking all of them can be very expensive.

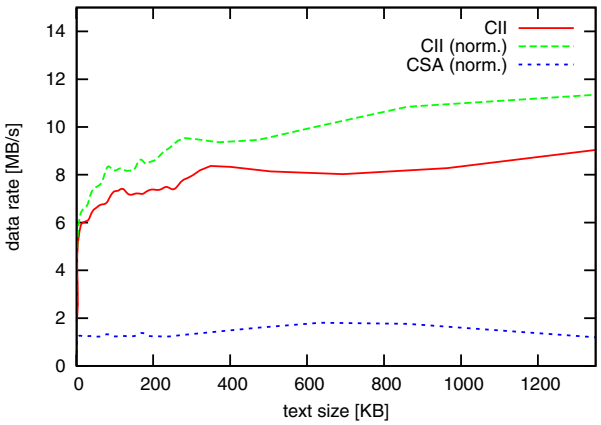


Figure 4: Document reporting speed. The curves are smoothed using gnuplot option `smooth bezier` in order to dampen fluctuations due to individual characteristics of documents.

Finally, Figure 4 shows the throughput for document reporting as a function of document size. Our scheme allows 6–8 MByte per second of text output. Retrieving data from disk (assuming an access latency of 5ms) would be faster beginning at around 32 KByte.[6] This suggests a hybrid storage scheme keeping longer files on disk. But note that in many applications, the texts are much shorter. For CSA, the bandwidth is about five times smaller.

In the appendix we also show results for the full wt2g data set which is about 5× larger. The average query time goes up proportionally to about 5ms for two term AND queries and about 70ms for two term phrase queries. Large query times are now 50ms for AND queries and 1.1 s for phrase queries. All these numbers are satisfactory for a text search engine, although it would be good to reduce the expensive phrase queries by indexing also some pairs and triples of word. The interesting question is how much space this would cost.

7 Conclusions and Future Work

A carefully engineered in-memory search engine based on inverted indexes and positional indexes allows fast queries using considerably less memory than the input size. We believe that we have not yet reached the end of what can be achieved with our approach: The bags of words are not optimally encodeable with Golomb codes and we could use more adaptive schemes. We could also compress the dictionary. With respect to query performance it seems most interesting to add a carefully selected set of phrases to the index in order to speed up the most expensive phrase queries. So far, space consumption and performance of index construction has not been our focus of attention. We will attack these issues together with the question of fast updates. The latter will be implemented using a small index for recent updates together with a batched update of the static data structures that should run in the background without replicating too much data. It would also be interesting to try suffix arrays that use the document IDs as their alphabet. Unfortunately, the current implementations seem to be tied to an 8 bit alphabet.

[6] On the first glance, it looks like parallel disks would be a way to mitigate disk access cost. However, with the advent of multicore processors, this option has become quite unattractive – one disk now costs more than a processing core so that even rack servers now tend to have less than one disk per processing core. Using blade servers, the ratio of cores to disks is even larger.

Acknowledgements We would like to thank Franz Färber and Holger Bast for valuable discussions.

References

[1] Sanders, P., Transier, F.: Intersection in integer inverted indices. In: Proceedings of the 9th Workshop on Algorithm Engineering and Experiments ALENEX, SIAM (2007) 71–83

[2] Golomb, S.: Run-length encodings. IEEE Transactions on Information Theory **12** (1966) 399–401

[3] Zobel, J., Moffat, A.: Inverted files for text search engines. ACM Computing Surveys **38** (2006)

[4] Culpepper, J.S., Moffat, A.: Compact set representation for information retrieval. In: SPIRE. Lecture Notes in Computer Science, Springer (2007)

[5] Zobel, J., Moffat, A., Sacks-Davis, R.: Storage management for files of dynamic records. In: Proceedings of the 4th Australian Database Conference, Brisbane, Australia, World Scientific (1993) 26–38

[6] Brown, E.W., Callan, J.P., Croft, W.B., Moss, J.E.B.: Supporting full-text information retrieval with a persistent object store. In: Proceedings of the Fourth Internationl Conference on Extending Database Technology EDBT'94. (1994) 365–378

[7] Strohman, T., Croft, W.B.: Efficient document retrieval in main memory. In: Proceedings of the 30th annual international conference on Research and development in information retrieval SIGIR '07, New York, NY, USA, ACM Press (2007) 175–182

[8] Luk, R.W.P., Lam, W.: Efficient in-memory extensible inverted file. Information Systems **32** (2007) 733–754

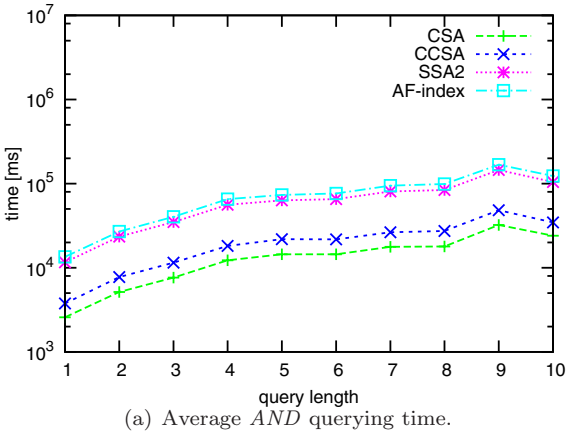
[9] Cutting, D., Pedersen, J.: Optimization for dynamic inverted index maintenance. In: Proceedings of the 13th annual international conference on Research and development in information retrieval SIGIR '90, New York, NY, USA, ACM Press (1990) 405–411

[10] Gutwin, C., Paynter, G., Witten, I., Nevill-Manning, C., Frank, E.: Improving browsing in digital libraries

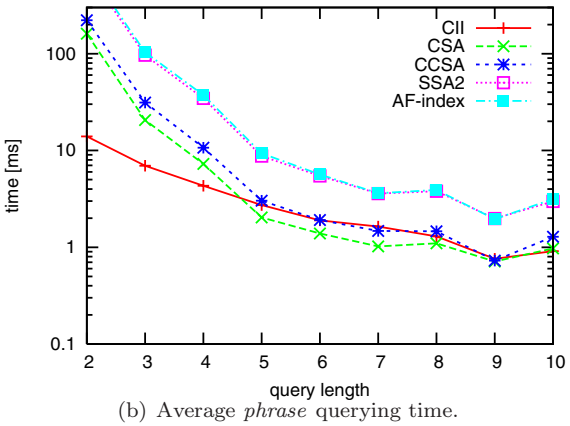
with keyphrase indexes. Decision Support Systems **27** (1999) 81–104
[11] Bahle, D., Williams, H.E., Zobel, J.: Efficient phrase querying with an auxiliary index. In: Proceedings of the 25th annual international conference on Research and development in information retrieval SIGIR '02, New York, NY, USA, ACM Press (2002) 215–221
[12] Williams, H.E., Zobel, J., Bahle, D.: Fast phrase querying with combined indexes. ACM Transactions on Information Systems **22** (2004) 573–594
[13] Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys **39** (2007)
[14] González, R., Navarro, G.: Compressed text indexes with fast locate. In: Proceedings of the 18th Annual Symposium on Combinatorial Pattern Matching CPM'07. Lecture Notes in Computer Science, Springer (2007) 216–227
[15] Witten, I.H., Moffat, A., Bell, T.C.: Managing Gigabytes: Compressing and Indexing Documents and Images. Morgan Kaufmann (1999)
[16] Ferragina, P., Fischer, J.: Suffix arrays on words. In: Proceedings of the 18th Annual Symposium on Combinatorial Pattern Matching CPM'07. Lecture Notes in Computer Science, Springer (2007) 328–339
[17] Puglisi, S.J., Smyth, W.F., Turpin, A.: Inverted files versus suffix arrays for locating patterns in primary memory. In: SPIRE. Volume 4209 of Lecture Notes in Computer Science., Springer (2006) 122–133
[18] Bast, H., Mortensen, C.W., Weber, I.: Output-sensitive autocompletion search. JIR (2007) to appear, special issue on SPIRE 2006.
[19] H. Bast et al.: Seminar: Searching with suffix arrays. http://search.mpi-inf.mpg.de/wiki/IrSeminarWs06 (2007)
[20] Baeza-Yates, R., Ribeiro-Neto, B.: Modern Information Retrieval. Addison-Wesley, New York (1999)
[21] Young, R.M.: Euler's constant. The Mathematical Gazette **75** (1991) 187–190
[22] Barbay, J., López-Ortiz, A., Lu, T.: Faster adaptive set intersections for text searching. In Àlvarez, C., Serna, M.J., eds.: Experimental Algorithms, 5th International Workshop, WEA. Volume 4007 of LNCS., Springer (2006) 146–157
[23] Jansen, B.J., Spink, A., Bateman, J., Saracevic, T.: Real life information retrieval: a study of user queries on the web. SIGIR Forum **32** (1998) 5–17

A Comparision of different Pizza&Chili index implementations on the WT2g subset

We have tried all implementations available on the Pizza&Chili site. However, some implementations crashed and others had so long indexing times (e.g., due to swapping) that we had to abort them. Table 2 summarizes the results of the other indices. Since CSA gives the best performance, we use it as our reference in the main paper.



(a) Average *AND* querying time.



(b) Average *phrase* querying time.

B Further experimental results for CII

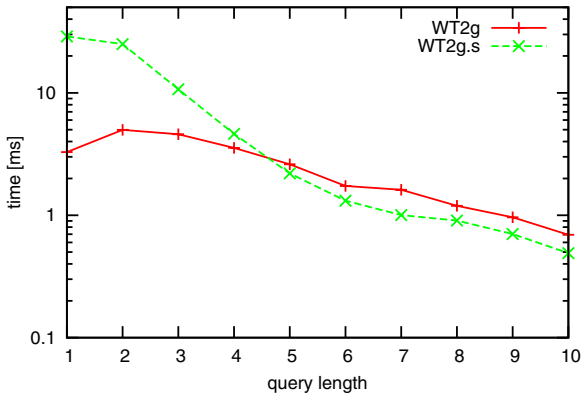
	WT2g	WT2g.s
# of documents	247 491	10 749 669
dictionary	85.9	55.3
document index	163.6	380.6
positional index	663.9	654.0
bag of words	139.4	410.2
text delta [MB]	557.6	-
sum [MB]	**1610.4**	**1500.2**
input size [MB]	2118.8	1487.0
compression	0.76	1.0[7]
indexing time [min]	**81.7**	**35.6**
peak mem usage [GB]	3.2	3.5

Table 3: Index properties

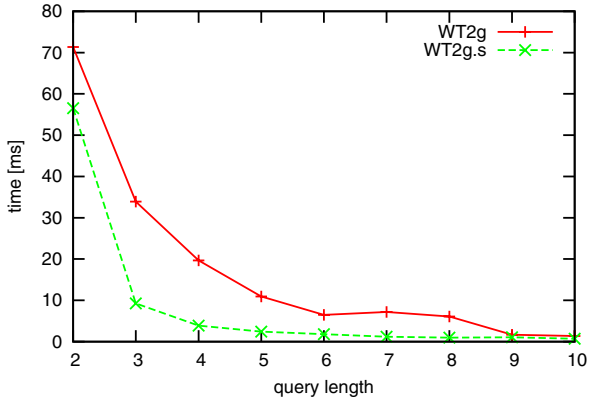
[7] Here, an average document length of 25.6 seems to be the limit of the compression potential of the bag-of-words approach. Obviously, the shorter the document lengths are, the more values have to be stored in the bags while indexing equal contents.

	CSA	CCSA	SSA2	AF-index
suffix array	230.8	500.8	302.5	279.2
doc bound list	0.1	0.1	0.1	0.1
sum [MB]	**230.9**	**500.9**	**302.6**	**279.3**
compression	0.64	1.39	0.84	0.77
indexing time [min]	**9.3**	**11.5**	**8.7**	**23.0**
peak mem usage [GB]	3.2	3.1	2.1	3.1

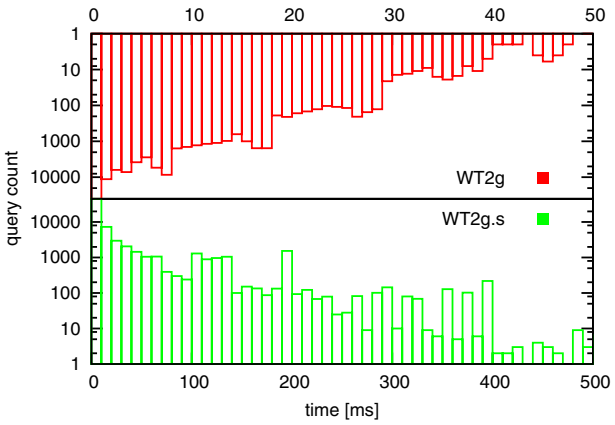
Table 2: Index properties of Pizza&Chili indexes.



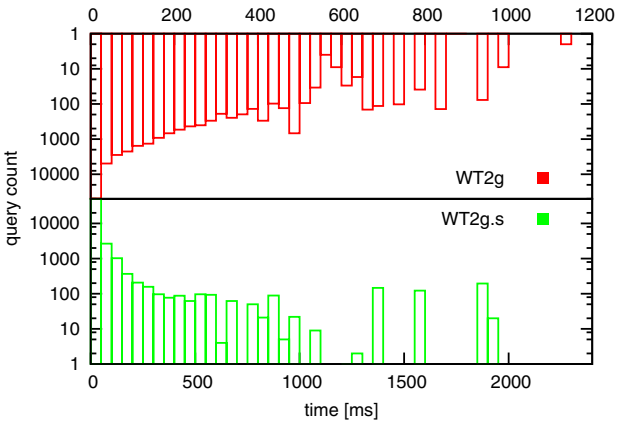
(c) Average *AND* querying time: AND queries for small query lenghts are slower on WT2g.s, as there are more documents and hence more results. Larger queries benefit from larger inverted lists, as in these cases more lookup lists are used.



(d) Average *phrase* querying time: Phrase queries are faster on WT2g.s because the position lists are shorter for small documents.



(e) Histogram of *AND* query running times.



(f) Histogram of *phrase* query running times.

Figure 5: Operations on CII